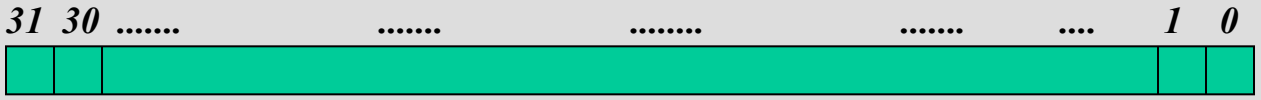
Lecture4(part1)

Topics covered:
Arithmetic

Number representation

Integers are represented as binary vectors

Suppose each word consists of 32 bits, labeled 0…31.



MSB (most significant bit) *LSB (least)*

Value of the binary vector interpreted as unsigned integer is:

$$V(b) = b_{31}.2^{31} + b_{30}.2^{30} + b_{29}.2^{29} + ... + b_1.2^1 + b_0.2^0$$

More generally in N bits,

$$V(b) = \sum_{n=0}^{n=N-1} b_n 2^n$$

Number representation (contd..)

- We need to represent both positive and negative integers.
- Three schemes are available for representing both positive and negative integers:
  - Sign and magnitude.
  - 1's complement.
  - 2's complement.
- All schemes use the Most Significant Bit (MSB) to carry the sign information:
  - If MSB = 0, bit vector represents a positive integer.
  - If MSB = 1, bit vector represents a negative integer.

Number representation (contd..)

- Sign and Magnitude:
  - Lower N-1 bits represent the magnitude of the integer
  - MSB is set to 0 or 1 to indicate positive or negative
- 1's complement:
  - Construct the corresponding positive integer (MSB = 0)
  - Bitwise complement this integer
- 2's complement:
  - Construct the 1's complement negative integer
  - Add 1 to this

Number representation (contd..)

B	Values represented		
$b_3\, b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Number representation (contd..)

Range of numbers that can be represented in N bits

Unsigned: $0 < V(b) < 2^N - 1$

Sign and magnitude: $-2^{N-1} - 1 < V(b) < 2^{N-1} - 1$

0 has both positive and negative representation

One's complement:: $-2^{N-1} - 1 < V(b) < 2^{N-1} - 1$

0 has both positive and negative representation

Two's complement:: $-2^{N-1} < V(b) < 2^{N-1} - 1$

0 has a single representation, easier to add/subtract.

Value of a bit string in 2's complement

How to determine the value of an integer given:

Integer occupies N bits.

2's complement system is in effect.

Binary vector b represents a negative integer, what is $V(b)$.

Write

$$b = 1\, b_{n-1}\, b_{n-2}\, b_{n-3} \ldots\ldots\ldots\ldots\ldots\ b_1\, b_0$$

Then

$$V(b) = -2^{n-1} + b_{n-2}\,2^{n-2} + b_{n-3}\,2^{n-3} + \ldots\ldots\ldots + b_2\,2^2 + b_1\,2^1 + b_0\,2^0$$

$$(v(b) = -2^{n-1} + b_{n-2}\,2^{n-2} + b_{n-3}\,2^{n-3} + \ldots\ldots\ldots + b_2\,2^2 + b_1\,2^1 + b_0\,2^0$$

showing negative and positive parts of the expression)

So, in 4 bits, 1011 is

$$v(1011) = -8 + 3 = -5$$

Addition of positive numbers

Add two one-bit numbers

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 1 \\ \hline 1\ 0 \end{array}$$

To add multiple bit numbers:

- Add bit pairs starting from the low-order or LSB (right end of bit vector)
- Propagate carries towards the high-order or MSB (left end of bit vector)

Addition and subtraction of signed numbers

- We need to add and subtract both positive and negative numbers.
- Recall the three schemes of number representation.
- Sign-and-magnitude scheme is the simplest representation, but it is the most awkward (inapplicable)for addition and subtraction operations. (has two forms for the zero number)
- 2's complement is the most efficient method for performing addition and subtraction of signed numbers.

Rules for addition and subtraction of signed numbers in 2's complement form

- To add two numbers:
  - Add their n-bit representations.
  - Ignore the carry out from MSB position.
  - Sum is the algebraically correct value in the 2's complement representation as long as the answer is in the range -2^{n-1} through $+2^{n-1}-1$.
- To subtract two numbers X and Y (X-Y):
  - Form the 2's complement of Y.
  - Add it to X using Rule 1.
  - Result is correct as long as the answer lies in the range -2^{n-1} through $+2^{n-1}-1$.

Addition Operations using Two's complement

The main advantage of using two's complement is converting subtraction into addition

(a)	0010	(+2)	(b)	0100	(+4)
	+ 0011	(+3)		+ 1010	(–6)
	0101	(+5)		1110	(–2)
(c)	1011	(–5)	(d)	0111	(+7)
	+ 1110	(–2)		+ 1101	(–3)
	1001	(–7)		0100	(+4)

Note: Don't forget to neglect the carry bit

Subtraction Operations using Two's complement

(e)	1101	(−3)	⇒	1101	
	− 1001	(−7)		+ 0111	
				0100	(+4)
(f)	0010	(+2)	⇒	0010	
	− 0100	(+4)		+ 1100	
				1110	(−2)
(g)	0110	(+6)	⇒	0110	
	− 0011	(+3)		+ 1101	
				0011	(+3)
(h)	1001	(−7)	⇒	1001	
	− 1011	(−5)		+ 0101	
				1110	(−2)
(i)	1001	(−7)	⇒	1001	
	− 0001	(+1)		+ 1111	
				1000	(−8)
(j)	0010	(+2)	⇒	0010	
	− 1101	(−3)		+ 0011	
				0101	(+5)

Overflow in integer arithemtic

- When the result of an arithmetic operation is outside the representable range an arithmetic overflow has occurred.
  - Range is -2^{n-1} through $+2^{n-1}-1$ for n-bit vector.
- When adding unsigned numbers, carry-out from the MSB position serves as the overflow indicator.
- When adding signed numbers, this does not work.
- Using 4-bit signed numbers, add +7 and +4:
  - Result is 1011, which represents –5.
- Using 4-bit signed integers, add –4 and –6:
  - Result is 0110, which represents +6.

```
  0111(+7)          1100(-4)
+ 0100(+4)        + 1010(-6)
------------      ------------
 1011(-5)  Overflow  0110(+6)  Overflow
```

Overflow in integer arithmetic (contd..)

- Overflow occurs when both the numbers have the same sign.
  - Addition of numbers with different signs cannot cause an overflow.
- Carry-out signal from the MSB (sign-bit) position is not a sufficient indicator of overflow when adding signed numbers.
- Detect overflow when adding X and Y:
  - Examine the signs of X and Y.
  - Examine the signs of the result S.
  - When X and Y have the same sign, and the sign of the result differs from the signs of X and Y, overflow has occurred.
  - How to detect overflow by a logical circuit?

Overflow =??????????

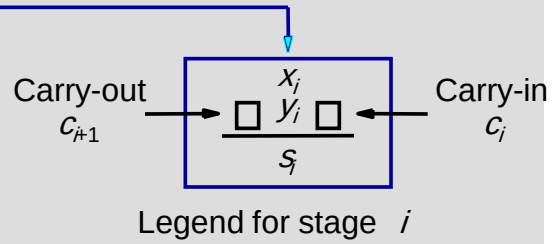
Addition/subtraction of signed numbers

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i\bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i\bar{y}_i\bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

Example:

$$\begin{array}{r} X \\ +\ Y \\ \hline Z \end{array} = \begin{array}{r} 7 \\ +\ 6 \\ \hline 13 \end{array} = \begin{array}{r} 0\ \ 1\ \ 1\ \ 1 \\ +\ 0\ 0\ 1\ 1\ 1\ 0\ 0 \\ \hline 1\ \ 1\ \ 0\ \ 1 \end{array}$$



At the i^{th} stage:

Input:

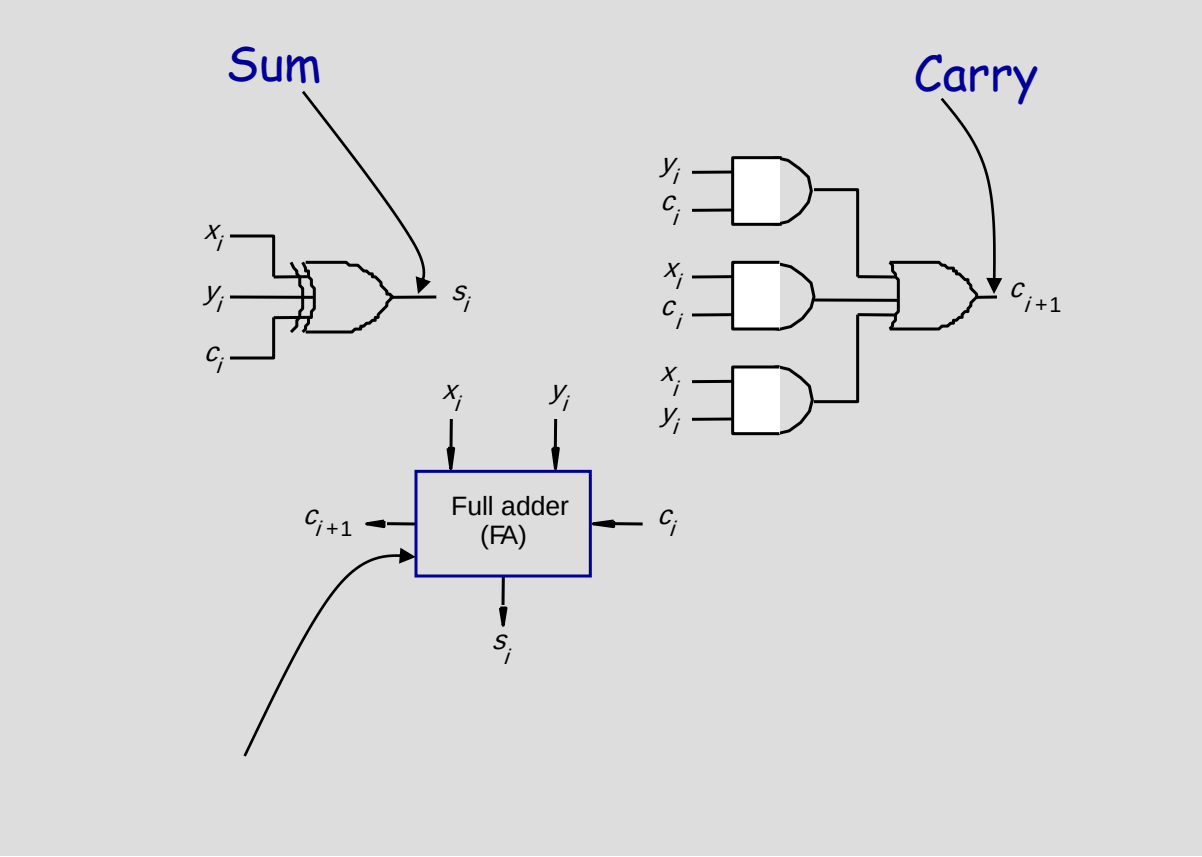
c_i is the carry-in

Output:

s_i is the sum

c_{i+1} carry-out to $(i+1)^{st}$ state

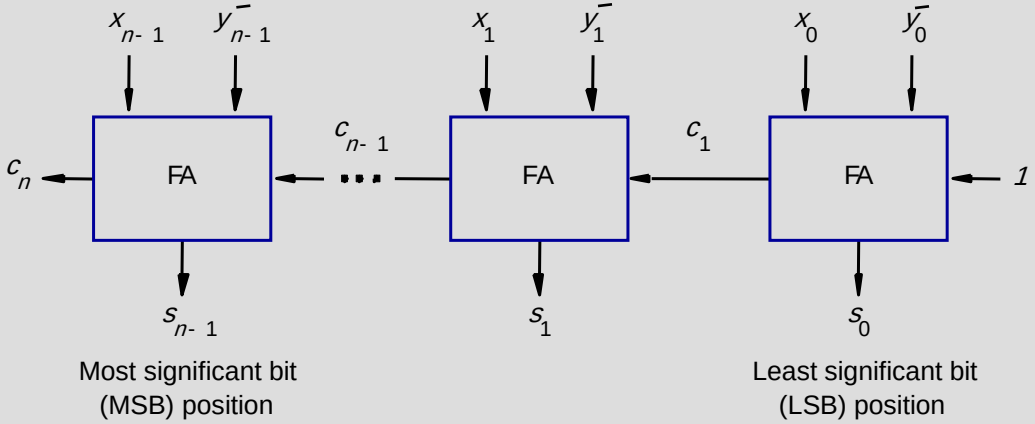
Addition logic for a single stage



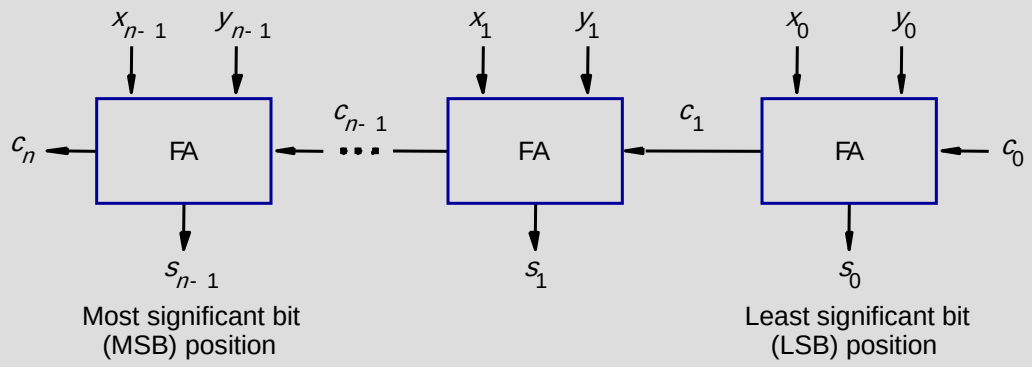
Full Adder (FA): Symbol for the complete circuit for a single stage of addition.

n-bit subtractor

- Recall $X - Y$ is equivalent to adding 2's complement of Y to X.
- 2's complement is equivalent to 1's complement + 1.
- $X - Y = X + \overline{Y} + 1$
- 2's complement of positive and negative numbers is computed similarly.

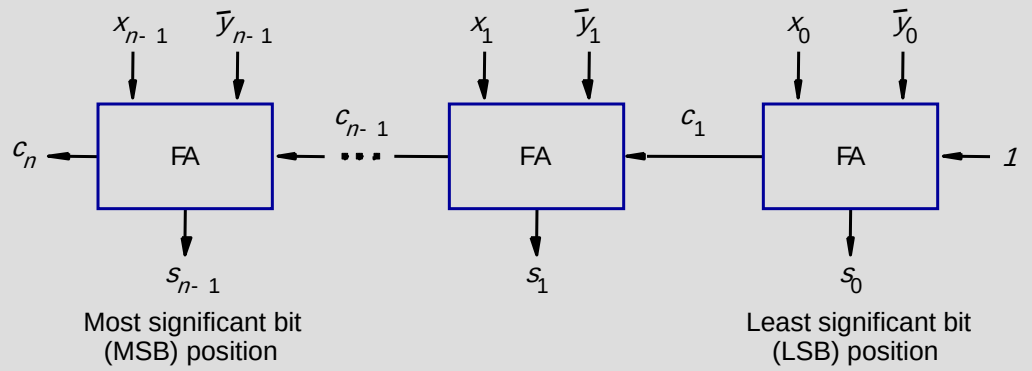


n-bit adder/subtractor



Adder inputs:

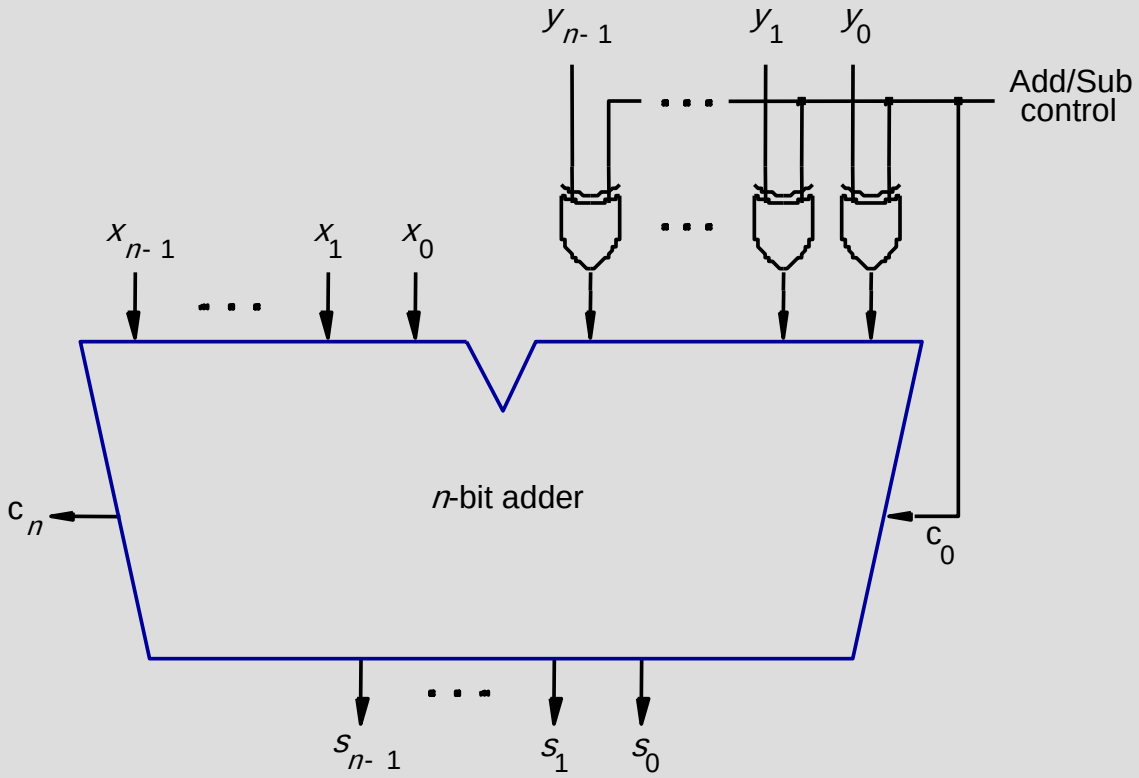
x_i, y_i, $c_o=0$



Subtractor inputs:

x_i, $\bar{y}_i$, $c_o=1$

n-bit adder/subtractor (contd..)



- Add/sub control = 0, addition.
- Add/sub control = 1, subtraction.

Detecting overflows

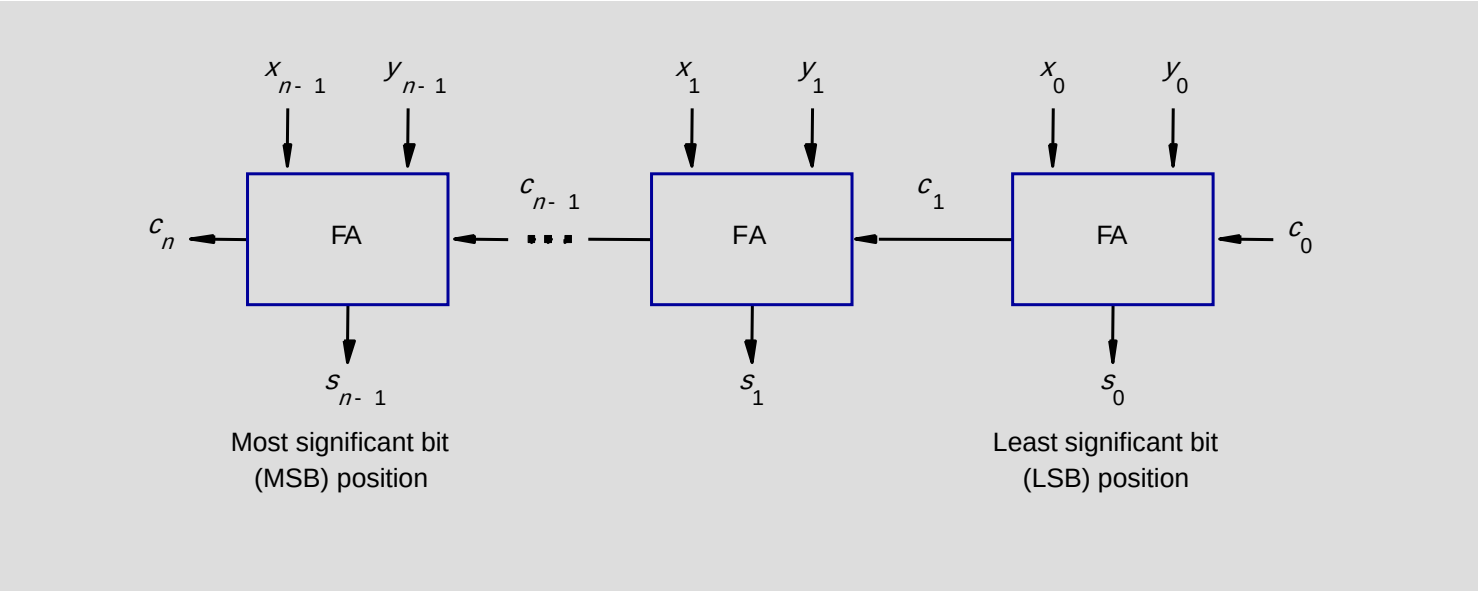
- Overflows can only occur when the sign of the two operands is the same.
- Overflow occurs if the sign of the result is different from the sign of the operands.
- Recall that the MSB represents the sign.
  - x_{n-1}, y_{n-1}, s_{n-1} represent the sign of operand x, operand y and result s respectively.
- Circuit to detect overflow can be implemented by the following logic expressions:

$$Overflow = x_{n-1} y_{n-1} \bar{s}_{n-1} + \bar{x}_{n-1} \bar{y}_{n-1} s_{n-1}$$

$$Overflow = c_n \oplus c_{n-1}$$

n-bit adder

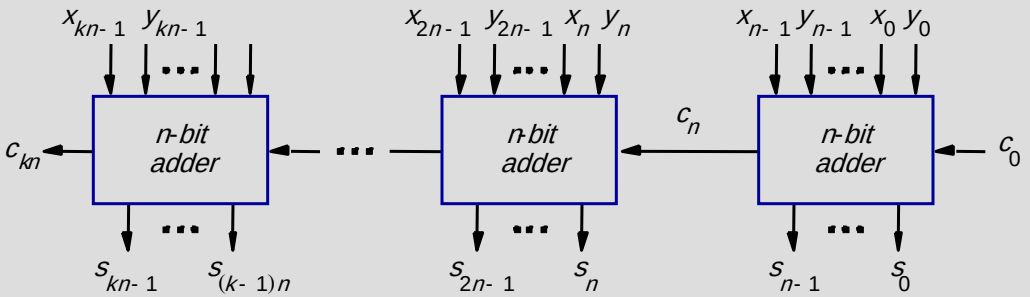
- Cascade n full adder (FA) blocks to form a n-bit adder.
- Carries propagate or ripple through this cascade, n-bit ripple carry adder.



Carry-in c_0 into the LSB position provides a convenient way to perform subtraction.

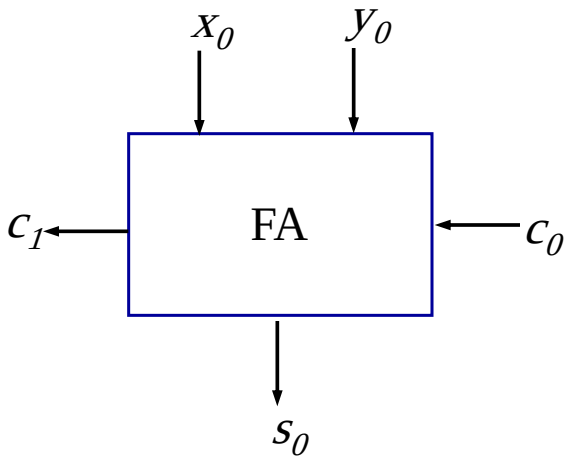
kn-bit adder

K n-bit numbers can be added by cascading *k n*-bit adders.



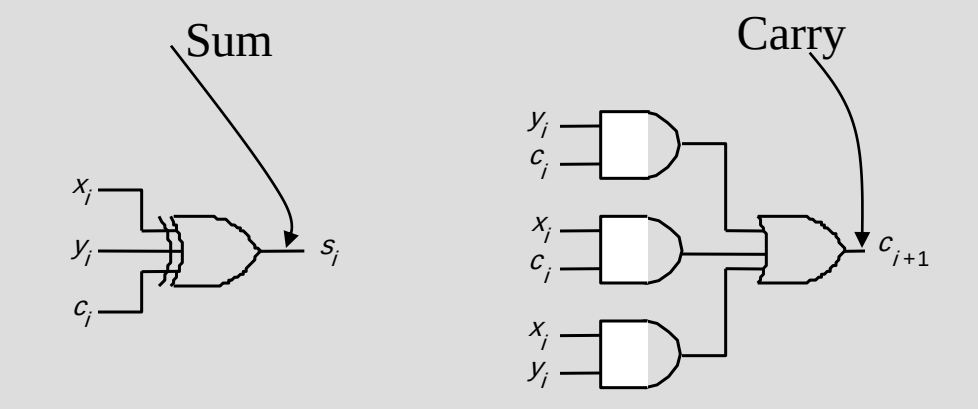
Each *n*-bit adder forms a block, so this is cascading of blocks.
Carries ripple or propagate through blocks, Blocked Ripple Carry Adder

Computing the add time



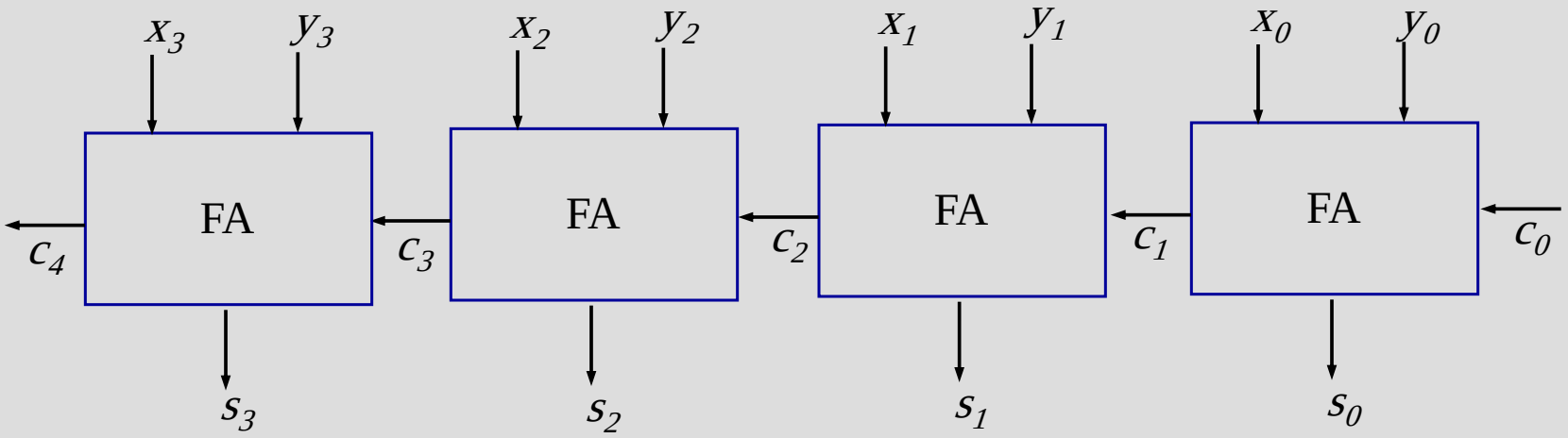
Consider 0^{th} stage:

- c_1 is available after 2 gate delays.
- s_1 is available after 1 gate delay.



Computing the add time (contd..)

Cascade of 4 Full Adders, or a 4-bit adder



- s_0 available after 1 gate delays, c_1 available after 2 gate delays.
- s_1 available after 3 gate delays, c_2 available after 4 gate delays.
- s_2 available after 5 gate delays, c_3 available after 6 gate delays.
- s_3 available after 7 gate delays, c_4 available after 8 gate delays.

For an n-bit adder, s_{n-1} is available after $2n-1$ gate delays
c_n is available after $2n$ gate delays.

Fast addition

Recall the equations:

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

Second equation can be written as:

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

We can write:

$$c_{i+1} = G_i + P_i c_i$$

$$where\, G_i = x_i y_i \; and \; P_i = x_i + y_i$$

- G_i is called generate function and P_i is called propagate function
- G_i and P_i are computed only from x_i and y_i and not c_i, thus they can be computed in one gate delay after X and Y are applied to the inputs of an n-bit adder.

Carry lookahead

$$c_{i+1} = G_i + P_i c_i$$

$$c_i = G_{i-1} + P_{i-1} c_{i-1}$$

$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1} c_{i-1})$$

$$continuing$$

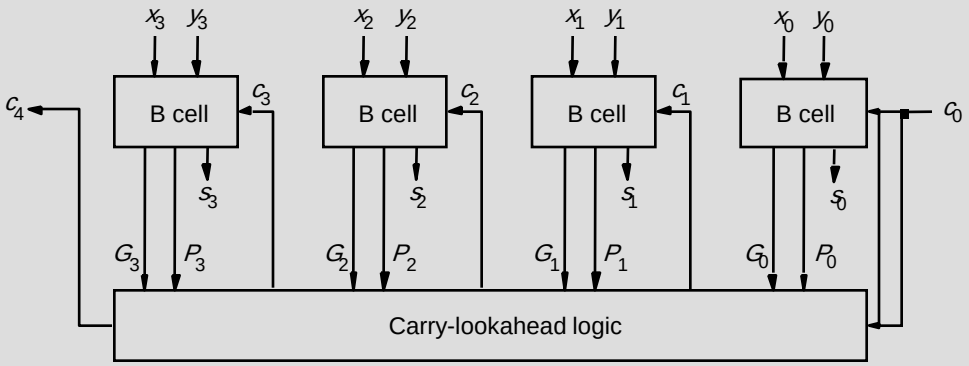
$$\Rightarrow c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1}(G_{i-2} + P_{i-2} c_{i-2}))$$

$$until$$

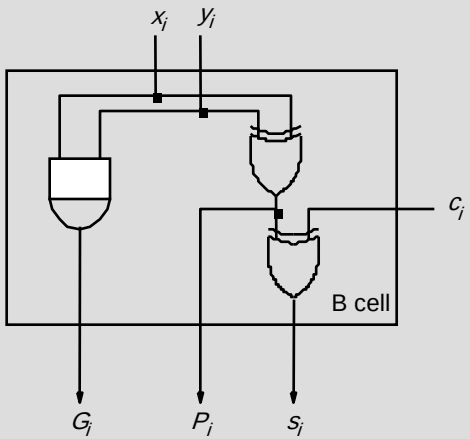
$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + .. + P_i P_{i-1} .. P_1 G_0 + P_i P_{i-1} ... P_0 c_0$$

- All carries can be obtained 3 gate delays after X, Y and c_0 are applied.
  - One gate delay for P_i and G_i
  - Two gate delays in the AND-OR circuit for c_{i+1}
- All sums can be obtained 1 gate delay after the carries are computed.
- Independent of n, n-bit addition requires only 4 gate delays.
- This is called Carry Lookahead adder.

Carry-lookahead adder



4-bit carry-lookahead adder.



B-cell for a single stage.

Carry lookahead adder (contd..)

- Performing n-bit addition in 4 gate delays independent of n is good only theoretically because of fan-in constraints.

$$c_{i+1} = G_i + PG_{i-1} + PP_{i-1}G_{i-2} + .. + PP_{i-1}..P_1G_0 + PP_{i-1}...P_0c_0$$

- Last AND gate and OR gate require a fan-in of (n+1) for a n-bit adder.
  - For a 4-bit adder ($n=4$) fan-in of 5 is required.
  - Practical limit for most gates.
- In order to add operands longer than 4 bits, we can cascade 4-bit Carry-Lookahead adders. Cascade of Carry-Lookahead adders is called Blocked Carry-Lookahead adder.

Blocked Carry-Lookahead adder

Carry-out from a 4-bit block can be given as:

$$c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$$

Rewrite this as:

$$P_0^I = P_3P_2P_1P_0$$

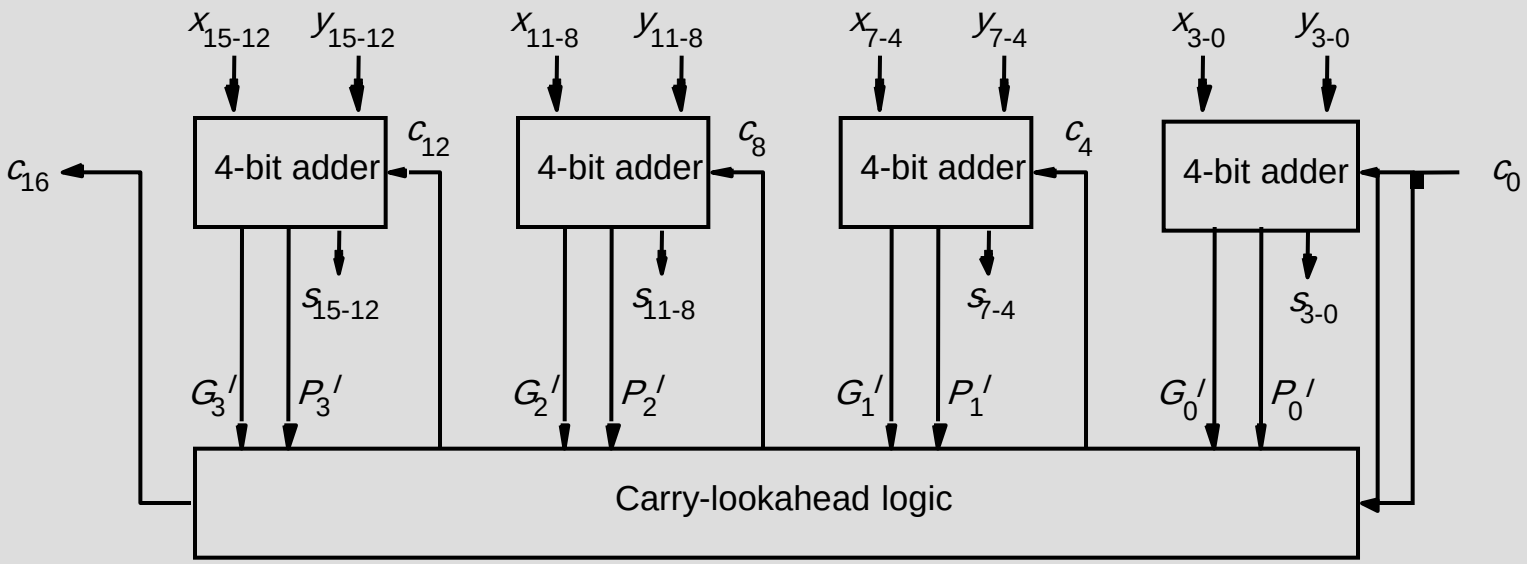
$$G_0^I = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$$

Subscript I denotes the blocked carry lookahead and identifies the block.

Cascade 4 *4*-bit adders, c_{16} can be expressed as:

$$c_{16} = G_3^I + P_3^I G_2^I + P_3^I P_2^I G_1^I + P_3^I P_2^I P_1^0 G_0^I + P_3^I P_2^I P_1^0 P_0^0 c_0$$

Blocked Carry-Lookahead adder (contd..)



After x_i, y_i and c_0 are applied as inputs:

- G_i and P_i for each stage are available after 1 gate delay.
- P^I is available after 2 and G^I after 3 gate delays.
- All carries are available after 5 gate delays.
- c_{16} is available after 5 gate delays.
- s_{15} which depends on c_{12} is available after 8 (5+3)gate delays (Recall that for a 4-bit carry lookahead adder, the last sum bit is available 3 gate delays after all inputs are available)

Multiplication of unsigned numbers

				1	1	0	1	(13) Multiplicand M
			x	1	0	1	1	(11) Multiplier Q
				1	1	0	1	Partial product (PP) #1
			1	1	0	1		Partial product (PP) #2
		0	0	0	0			Partial product (PP) #3
	1	1	0	1				Partial product (PP) #4
1	0	0	0	1	1	1	1	(143) Product P

- Product of 2 n-bit numbers is at most a $2n$-bit number.
- We should expect to store a double-length result.

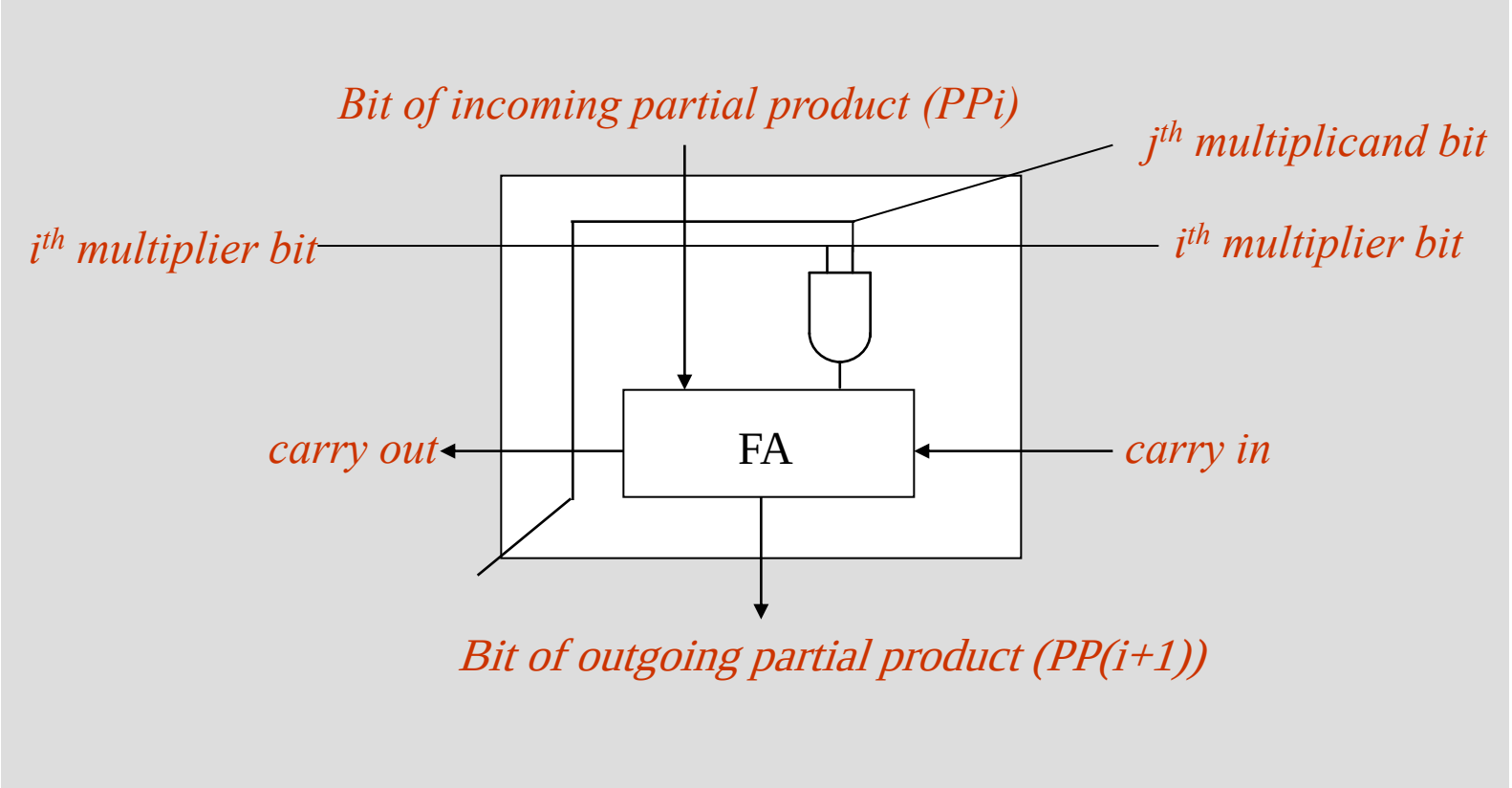
Unsigned multiplication can be viewed as addition of shifted versions of the multiplicand.

Multiplication of unsigned numbers (contd..)

- We added the partial products at end.
  - Alternative would be to add the partial products at each stage.
- Rules to implement multiplication are:
  - If the i^{th} bit of the multiplier is 1, shift the multiplicand and add the shifted multiplicand to the current value of the partial product.
  - Hand over the partial product to the next stage
  - Value of the partial product at the start stage is 0.

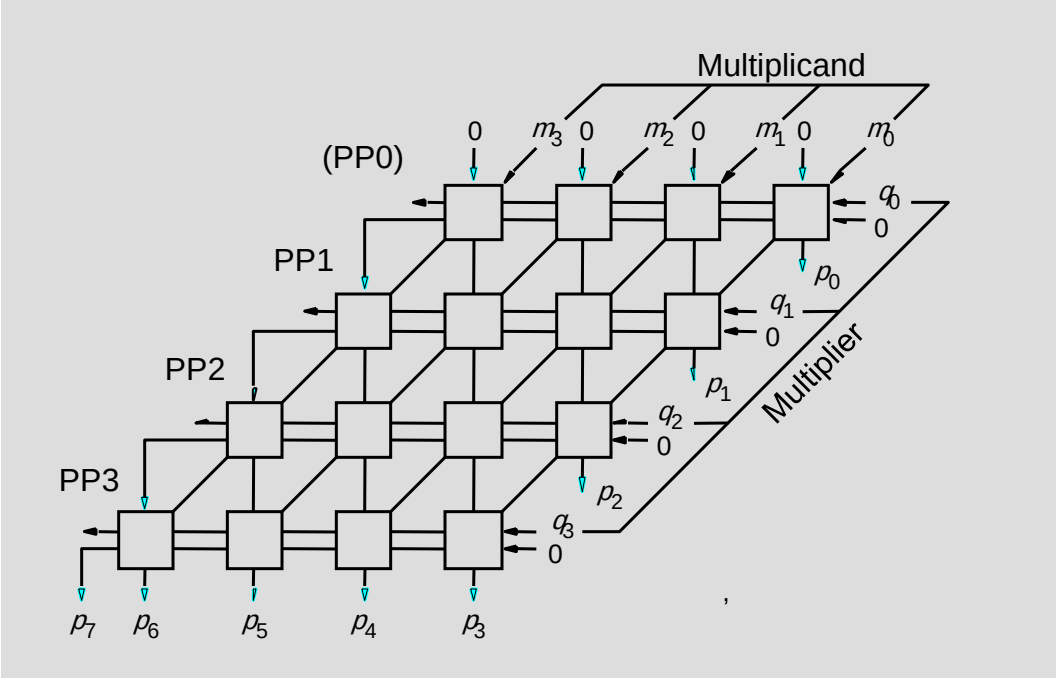
Multiplication of unsigned numbers (contd..)

Typical multiplication cell



Combinatorial array multiplier

Combinatorial array multiplier



Product is: $p_7 p_6 .. p_0$

Multiplicand is shifted by displacing it through an array of adders.

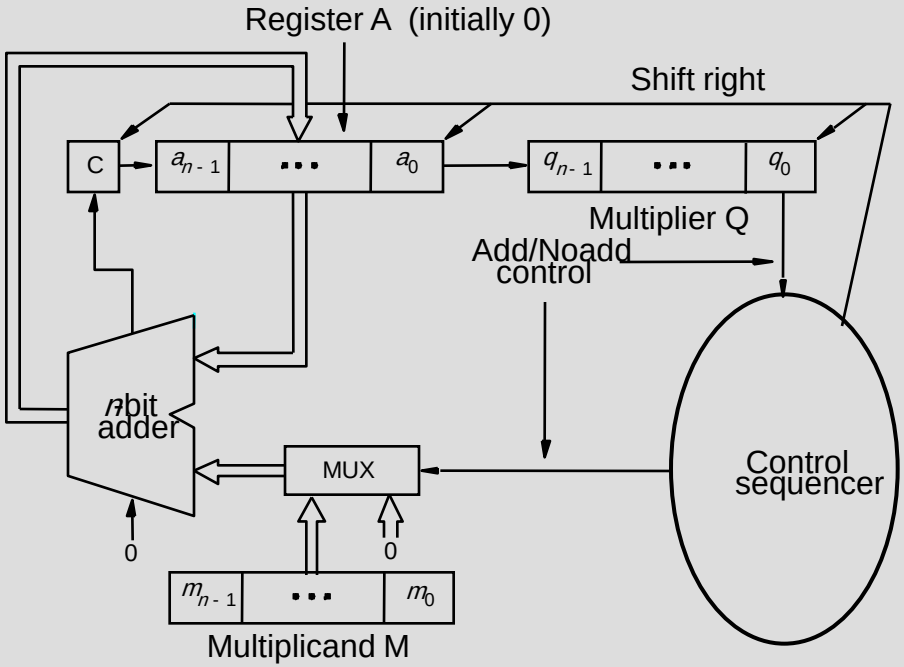
Combinatorial array multiplier (contd..)

- Combinatorial array multipliers are:
  - Extremely inefficient.
  - Have a high gate count for multiplying numbers of practical size such as 32-bit or 64-bit numbers.
  - Perform only one function, namely, unsigned integer product.
- Improve gate efficiency by using a mixture of combinatorial array techniques and sequential techniques requiring less combinational logic.

Sequential multiplication

- Recall the rule for generating partial products:
  - If the ith bit of the multiplier is 1, add the appropriately shifted multiplicand to the current partial product.
  - Multiplicand has been shifted left when added to the partial product.
- However, adding a left-shifted multiplicand to an unshifted partial product is equivalent to adding an unshifted multiplicand to a right-shifted partial product.

Sequential multiplication (contd..)



- Load Register A with 0.
- Registers are used to store multiplier and multiplicand.
- Each cycle repeat the following steps:

1. If the LSB $q_0=1$:
   - Add the multiplicand to A.
   - Store carry-out in flip-flop C

   Else if $q_0 = 0$
   - Do not add.
2. Shift the contents of register A and Q to the right, and discard q_0.

Sequential multiplication (contd..)